**Spring 入门**

# Spring是什么

1. Spring 是一个开源框架.
2. Spring 为简化企业级应用开发而生. 使用 Spring 可以使简单的 JavaBean 实现以前只有 EJB 才能实现的功能.
3. Spring 是一个 IOC(DI) 和 AOP 容器框架.

# Spring的作用

1）用来整合其他框架，可以让程序的结构更为松散，实现低耦合

2）简化事务控制，实现声明式事务(相对于编程式事务来言)

# 具体描述

具体描述 Spring:

**轻量级**：**Spring 是非侵入性的** - 基于 Spring 开发的应用中的对象可以不依赖于 Spring 的 API

**依赖注入**(DI --- dependency injection、IOC)

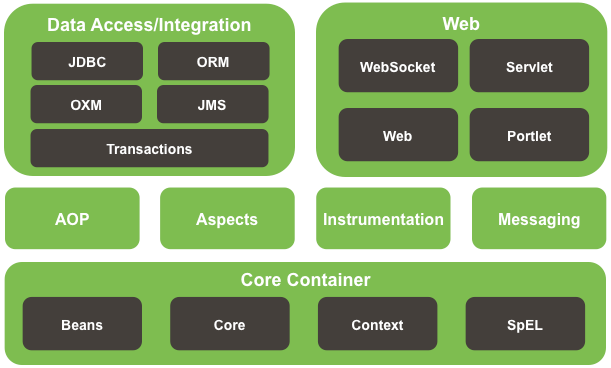
**面向切面编程**(AOP --- aspect oriented programming)

**容器**: Spring 是一个容器, 因为它包含并且管理应用对象的生命周期

**框架**: Spring 实现了使用简单的组件配置组合成一个复杂的应用. 在 Spring 中可以使用 XML 和 Java 注解组合这些对象

**一站式**：在 IOC 和 AOP 的基础上可以整合各种企业应用的开源框架和优秀的第三方类库 （实际上 Spring 自身也提供了展现层的 SpringMVC 和 持久层的 Spring JDBC）

# Spring模块组成



# 两个核心思想：

## IOC&DI：

IOC控制反转 Inversion of Control

所谓的 IOC：是指将某一类对象的控制权（对象创建和方法调用）交给外部容器来负责。

DI：依赖注入 Dependency Inject

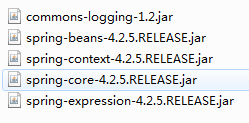
所谓 DI：对象依赖的属性，由容器给此属性赋值的过程

## AOP

aop：面向切面编程

# 第一个spring程序

## 添加spring核心依赖库

**

## 创建spring的全局配置文件applicationContext.xml

在resouces目录下新建spring的配置文件applicationContext.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

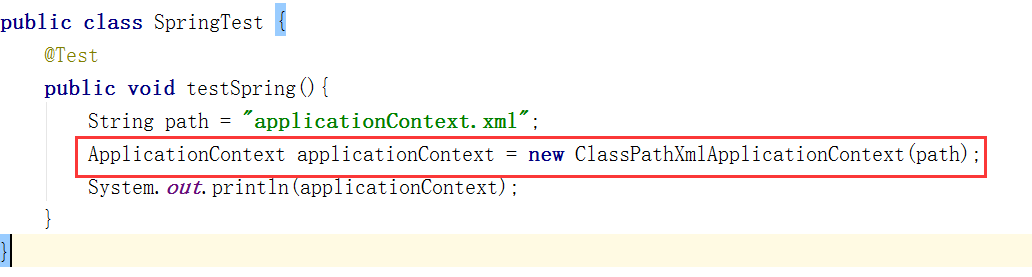
xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

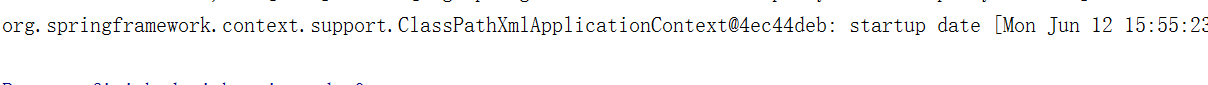
*http://www.springframework.org/schema/beans/spring-beans-4.2.xsd"*>

</beans>

## 编写测试类，测试spring环境是否搭建成功



若环境成功搭建，数据结果为非空，如下：



# IOC控制反转

在spring之前，我们程序中需要一个对象实例是直接通过new的方式来创建，或者通过对象工厂的方式来创建，而在spring中，是通过IOC控制反转的方式来创建实例，简单来说就是不用显示的new对象，而是将对象的创建交给spring来实现。这样对象创建的控制权就由应用内转向了应用外，由此形成控制反转。

## 建立一个类Human。

**public** **class** Human {

**public** **void** say() {

System.***out***.println("这个人正在与人交谈...");

}

}

## 在applicationContext.xml中配置该类的实例

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-4.2.xsd"*>

<bean id=*"human"* class=*"spring1.Human"*></bean>

</beans>

PS：通过bean标签来定义实例，class属性指定类的全限定类名，id为唯一标识，在整个应用内id应该唯一。

## **在测试类中，我们通过spring的控制反转来获取一个该类的实例，并调用该实例的sa**y方法

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ApplicationContext applicationContext = **new**

ClassPathXmlApplicationContext("applicationContext.xml");

Human human =(Human) applicationContext.getBean("human");

human.say();

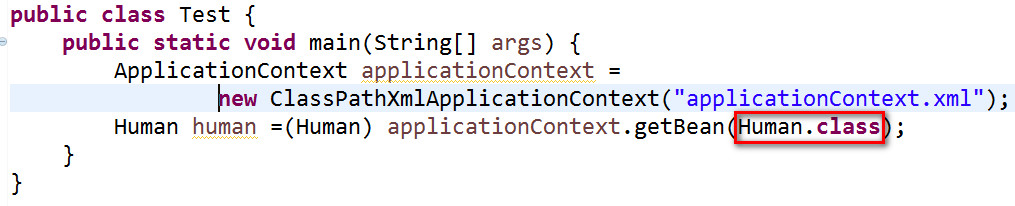
}

}

由此可见，使用spring后，对象的创建不再是由程序内部来控制，而是交给spring容器来管理。

PS：在spring'中，从容器中获取对象的实例的方式有以下两种方式：

1. 通过bean的id属性来获取，以上的例子就是采用此方式
2. 通过实例的类类型来获取，如下例子所示



## bean的作用域（重点）

spring中的bean默认为单例模式，即spring容器创建的bean在整个应用声明周期内，仅有一个实例，若需要修改为非单例模式可以通过为bean的scope属性指定值**prototype**实现。

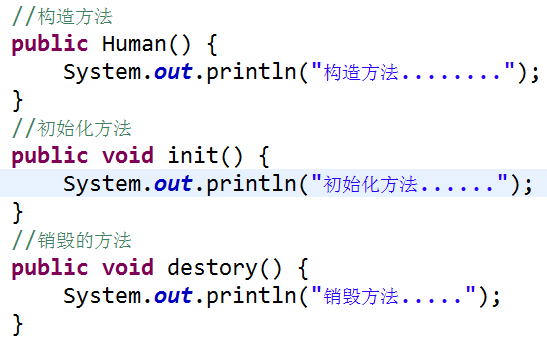
C:\Users\Administrator\Desktop\2018-08-11_092034.jpg

## bean的生命周期

生命周期指bean实例从创建到销毁的整个过程，通过spring提供的回调方法实现。定义bean时指定两个回调方法，控制bean的初始化和销毁时的动作。

1. Init-method：定义初始化的时候的回调方法，该方法会在实例初始化完成后及时执行，即在构造方法之后执行；
2. Destory-mehtod：定义对象销毁时的回调方法，该方法会在实例被销毁的时候执行。

### Human.java



### applicationContext.xml

<bean id="human" class="spring1.Human"

init-method="init" destroy-method="destory">

</bean>

### 测试代码

**public** **static** **void** main(String[] args) {

ApplicationContext applicationContext =

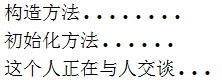
**new** ClassPathXmlApplicationContext("applicationContext.xml");

Human human =(Human) applicationContext.getBean(Human.**class**);

human.say();

}

### 结果



### 注意

销毁的方法仅针对单例模式起作用，对于非单例模式无效。

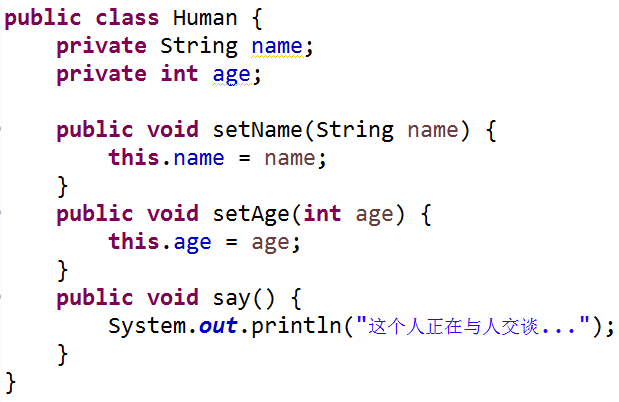
# DI依赖注入

在spring之前，我们要为对象实例的属性设置值，一般是通过

1. 为属性提供setter方法，然后通过对象实例的setter方法设置值；
2. 在程序中，通过构造方法为对象的属性设置值

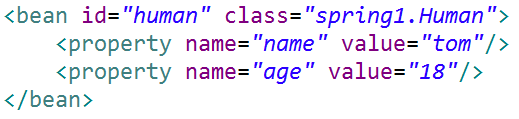
但是不管采用哪种方式，均是在应用内为属性设置值，程序深度耦合，而在spring中可以将该工作交由spring来完成，让程序在运行期，由外部容器(spring)动态地将依赖对象注入到组件中

## 为前面例子Human类定义属性并提供setter方法



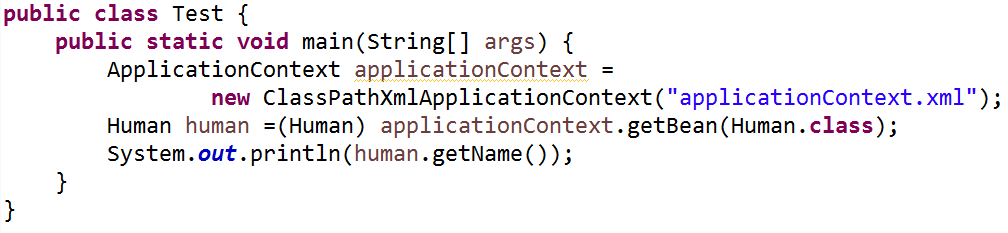
PS：为了方便获取数据，通常情况也会生成属性的getter方法

## 在spring配置文件applicationContext.xml中通过控制反转为name和age属性注入值。



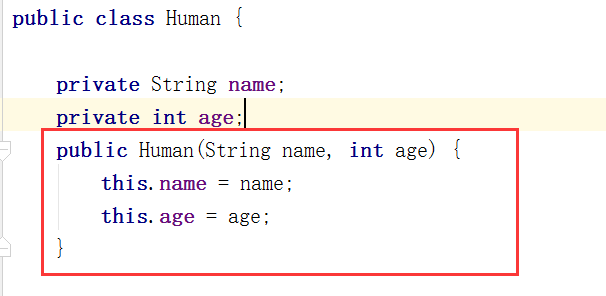
PS:spring中通过bean的子标签properties为属性注入值，其中name为Human类中定义的属性，value为具体注入的值。

## 测试类

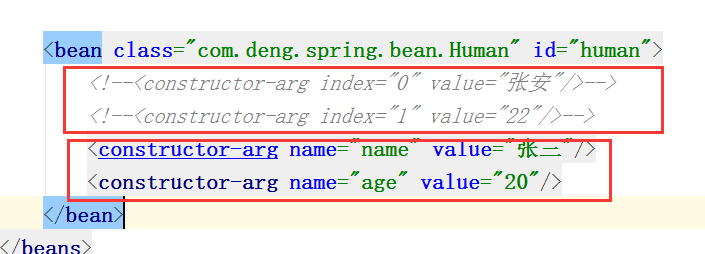


## PS：在spring中，bean的属性注入有2种方式：

1. 基于setter方法注入，上面的实例就是通过setter方法注入
2. 基于构造方法注入，下面的实例为通过构造方法注入
   1. 为human类定义带参数的构造方法



* 1. 在spring配置文件中通过构造方法来注入属性（了解）



上面的方式为基于构造方法的参数的索引位置实现注入，下面的是通过参数名的方式进行注入，推荐第二种方式。

## 为bean属性注入不同类型的值

### String类型，普通数据类型

1. 在类中定义待注入的属性
2. 为每个属性提供setter方法
3. 在bean中通过<property name="name" value="lily"/>注入

### 引用数据类型（对象）

1. 创建Student类

public class Student {

private String sname;

private int score;

}

1. 在Human中添加Student属性并提供setter方法

**public** **class** Human {

**private** String name;

**private** **int** age;

//定义“学生”属性

**private** Student student;

}

1. 在spring中通过bean为Student配置对象

<!-- 配置Student对象 -->

<bean class=*"spring1.Student"* id=*"student"*>

<property name=*"sname"* value=*"小张同学"*/>

<property name=*"score"* value=*"80"*/>

</bean>

1. 通过ref应用对应的资源

<!-- 配置Human对象 -->

<bean id=*"human"* class=*"spring1.Human"*>

<property name=*"name"* value=*"tom"*/>

<property name=*"age"* value=*"18"*/>

<!-- 引入学生对象 -->

<property name=*"student"* ref=*"student"*/>

</bean>

### 集合注入

spring中处理的集合包含：List，Set，Map，Properties，数组

Human类中定义集合类型属性，以及对应的setter方法

//集合属性

**private** List list;

**private** Set set;

**private** Map map;

**private** Properties properties;

**private** String[] arrs;

spring配置文件中注入方式如下：

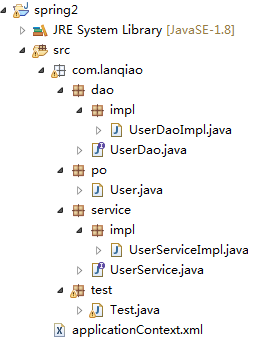
<**property name="list"**>  
 <**list**>  
 <**value**>1</**value**>  
 <**value**>2</**value**>  
 <**value**>4</**value**>  
 <**value**>5</**value**>  
 </**list**>  
</**property**>  
<**property name="set"**>  
 <**set**>  
 <**value**>a</**value**>  
 <**value**>b</**value**>  
 <**value**>c</**value**>  
 </**set**>  
</**property**>  
<**property name="map"**>  
 <**map**>  
 <**entry**>  
 <**key**>  
 <**value**>key1</**value**>  
 </**key**>  
 <**value**>value1</**value**>  
 </**entry**>  
 <**entry**>  
 <**key**>  
 <**value**>key2</**value**>  
 </**key**>  
 <**value**>value2</**value**>  
 </**entry**>  
 </**map**>  
</**property**>  
<**property name="properties"**>  
 <**props**>  
 <**prop key="k1"**>v1</**prop**>  
 <**prop key="k2"**>v2</**prop**>  
 </**props**>  
</**property**>

<**property name="arrs"** >  
 <**array**>  
 <**value**>str1</**value**>  
 <**value**>str2</**value**>  
 <**value**>str3</**value**>  
 </**array**>  
</**property**>

PS：若集合元素是一个应用类型，则将<value>变更为<ref bean=**"**被注入的对象的名称**"**/>

# 案例：使用spring配置三层结构

## 目录结构



**PS：没有servlet层，把test当成serlvet处理**

## java代码

### po类

**public** **class** User {

**private** String name;

**private** **int** sex;

}

### dao层

#### UserDao接口

**public** **interface** UserDao {

//查询所有用户

**void** findAllUser();

}

#### UserDaoImpl实现类

**public** **class** UserDaoImpl **implements** UserDao {

@Override

**public** **void** findAllUser() {

System.***out***.println("Dao层执行了........");

}

}

### serivce层

### UserService接口

**public** **interface** UserService {

//查询所有用户

**void** findAllUser();

}

### UserServiceImpl实现类

**public** **class** UserServiceImpl **implements** UserService {

**private** UserDao userDao;

@Override

**public** **void** findAllUser() {

System.***out***.println("service层执行了...");

userDao.findAllUser();

}

**public** UserDao getUserDao() {

**return** userDao;

}

**public** **void** setUserDao(UserDao userDao) {

**this**.userDao = userDao;

}

}

### Test类

**public** **class** Test {

**public** **static** **void** main(String[] args) {

ApplicationContext ac = **new** ClassPathXmlApplicationContext("applicationContext.xml");

UserService userService = (UserService) ac.getBean("userService");

userService.findAllUser();

}

}

## spring配置管理

<!-- 配置dao层 -->

<bean id=*"userDao"* class=*"com.lanqiao.dao.impl.UserDaoImpl"*/>

<!-- 配置service层 -->

<bean id=*"userService"*

class=*"com.lanqiao.service.impl.UserServiceImpl"*>

<property name=*"userDao"* ref=*"userDao"*/>

</bean>